Wrocław, 24 maja 2020

Michał Ryśkiewicz, 241383
Termin: WT/TN/7:30

Ocena:

Sprawozdanie z laboratorium nr 4 z przedmiotu

„Organizacja i Architektura Komputerów”

Rok akademicki 2019/2020, kierunek: INF

Prowadzący:
Mgr inż. Tomasz Serafin

1. Cel ćwiczenia

Zadaniem realizowanym w trakcie zajęć było opracowanie programu wykonującego podstawowe operacje arytmetyczne (dodawanie, odejmowanie, mnożenie i dzielenie) na wektorach 128 bitowych. Dodatkowo operacje te miały być wykonywane z wykorzystaniem typu obliczeń SIMD (Single Instruction, Multiple Data), oraz SISD (Single Instruction, Single Data).W programie zgodnie z wytycznymi, znalazły się również metody odpowiedzialne za pomiar czasu poszczególnych funkcji, fragmenty kodu odpowiedzialne za zapis pomiarów do pliku, a także generator liczb pseudolosowych o zadanej wielkości.

Realizowane zadanie zostało wykonane z wykorzystaniem języka C, wraz z umieszczonymi w nim fragmentami kodu języka Assemblera.

2. Przebieg pracy nad programem

Pierwszym etapem jakim należało zająć się w programie (po wykonaniu rozeznania teoretycznego) była implementacja odpowiedniej struktury przechowującej wektor w postaci czterech zmiennych typu „float". Zdecydowałem się na użycie tej zmiennej ze względu na dobre zrozumienie operacji jakie wykonywane są na liczbach zmiennoprzecinkowych zdobytych w trakcie laboratoriów nr. 3.

Następnym bardziej istotnym krokiem było przygotowanie odpowiedniej funkcji generującej zmienne. Przygotowane przeze mnie funkcje noszą nazwy „initializeNumberToTest" i „initializeRandomNumbers". Pierwsza z nich przyjmuje jako argument odpowiednio zadaną liczbę typu „float", a następnie przypisuje ją do zmiennych wchodzących w skład danego wektora. Funkcja ta była o tyle istotna, ponieważ dzięki niej można było w szybki sposób sprawdzać powstałe błędy w programie. Natomiast druga z wymienionych funkcji generowała losowe wartości liczby typy „float" i przypisywała je do odpowiednich składowych danych wektorów.

Następną istotną rzeczą, która została wykonana było napisanie funkcji wykonujących zadane operacje arytmetyczne w typie obliczeń SIMD. Argumenty wszystkich funkcji przyjmowały dwa wektory na których należało wykonać odpowiednie działania (przekazane przez wartość w celu uniknięcia ew. nadpisania lub zmiany zmiennych składających się w danej strukturze) oraz wskaźnika do wektora przechowującego wynik. Ostatni z argumentów postanowiłem przekazać w postaci wskaźnika, ponieważ nie istniało ryzyko nadpisania lub uszkodzenia danych, ponieważ wektor ten jedynie przechowywał wynik poszczególnych operacji. Następnie zgodnie z informacjami zawartymi w literaturze[1] należało zdefiniować odpowiednie operacje w języku assemblera. Szczegółowy opis danych funkcji został przedstawiony w rozdziale nr 4.

Kolejnym krokiem było napisanie odpowiednich funkcji wykonujących operacje arytmetyczne w typie obliczeń SISD. Analogicznie jak w powyższym przykładzie, wszystkie funkcje przyjmowały te same argumenty (z tych samych powodów), a bardziej szczegółowy opis został przedstawiony w rozdziale nr 4.

Ostatnim z istotnych kroków wykonanych w programie było napisanie odpowiednich funkcji odpowiedzialnych za wykonanie wszystkich dostępnych operacji arytmetycznych, zmierzenie czasu ich wykonywania w μs, a następnie obliczeniu średniej czasu działania każdej z nich.

3. Napotkane problemy

Główne problemy na jakie natknąłem się w trakcie realizacji zadania to m.in. połączenie fragmentów kodu języka assemblera z językiem C i mierzenie czasu w języku C.

Pierwszy z wymienionych problemów, należało rozwiązać poprzez odpowiednie zaznajomienie się z informacjami dostępnymi w literaturze. Szczególnie istotne okazało się tutaj zrozumienie wykonywania obliczeń przy wykorzystaniu instrukcji SSE[2], a także w jaki sposób korzysta się z rozszerzeń języka assembler w języku C. W ostatnim z wymienionych istotna okazała się informacja o kolejności pisania kodu, tj. w pierwszych liniach kodu powinien znaleźć się kod w języku assemblera, następnie po dwukropku znajdują się odpowiednie operacje dot. danych wyjściowych, a po następnym dwukropku operacje dot. danych wejściowych do funkcji.

Następnym z problemów na jaki natknąłem się w trakcie implementacji programu, był pomiar czasu działania funkcji w języku C. O ile sama procedura pomiarów nie była skomplikowana, tak trudnością okazały się początkowo otrzymywane wyniki. Przez wykorzystanie makra CLOCKS_PER_SEC, otrzymane wyniki dla wszystkich obliczeń wynosiły 0.000001s przez co analiza porównawcza wykorzystanych typów obliczeń okazałaby się niemiarodajna.

Rozwiązaniem tego problemu okazało się stworzenie makra „microseconds", które skalowało czas, tak by otrzymane wyniki zapisywane były w mikro sekundach. Dzięki temu otrzymywałem faktyczne różnice w czasach działania poszczególnych funkcji, co pozwoliło mi na późniejszą analizę porównawczą.

4. Kluczowe fragmenty kodu

Pierwszą z przedstawionych funkcji, jest ta wykonująca operację dodawania w typie SIMD. Listing jej kodu prezentuję poniżej.

```
1.  void addSIMD(struct vector firstVector, struct vector secondVector,struct  vector* resultVector){
2.      __asm__(
3.          "movaps %1, %%xmm0;"
4.          "movaps %2, %%xmm1;"
5.          "addps %%xmm1, %%xmm0;"
6.          "movaps %%xmm0, %0;"
7.          : "=m" (*resultVector)
8.          : "m" (firstVector),"m" (secondVector)
9.          );
10. };
```

Listing 1. Funkcja dodająca dwa wektory w typie obliczeń SIMD.

W funkcji tej wykonywane są następujące operacje. W 3 linii funkcji, zmienna float pochodząca z pierwszego wektora umieszczana jest w rejestrze xmm0. Analogiczna sytuacja dla zmiennej pochodzącej z drugiego wektora ma miejsce w 4 linii listingu, ale zmienna ta umieszczana jest w rejestrze xmm1. Następnie zmienne te są dodawane a wynik umieszczany jest w rejestrze xmm0. W linii nr 6 następuje przeniesienie wyniku z rejestru xmm0 do zmiennej wektora wynikowego. Przypisanie „%1" i „%2" do zmiennych pochodzących z wektorów wejściowych następuje w linii 8, natomiast przypisanie „%0" do zmiennej wynikowej następuje w linii 7. Warto nadmienić dodatkowo, iż oznaczenie „m"

oznacza, że zmienna ta przechowywana jest w pamięci (nie w rejestrze), natomiast oznaczenie „=m” oznacza, że zmienna ta jest dostępna tylko w trybie zapisu.

Drugą istotną funkcją którą, pragnę przedstawić jest ta odpowiedzialna za wykonywanie operacji dodawania w typie SISD. Kod tej funkcji prezentuję poniżej.

```
1.  void addSISD(struct vector firstVector, struct vector secondVector,struct  vector* resultVector ){
2.      __asm__(
3.          // adding x0
4.          "fld %4;"
5.          "fadd %8;"
6.          "fstp %0;"
7.
8.          // adding x1
9.          "fld %5;"
10.         "fadd %9;"
11.         "fstp %1;"
12.
13.         //adding x2
14.         "fld %6;"
15.         "fadd %10;"
16.         "fstp %2;"
17.
18.         //adding x3
19.         "fld %7;"
20.         "fadd %11;"
21.         "fstp %3;"
22.
23.         :
24.           "=m"((*resultVector).x0),
25.           "=m"((*resultVector).x1),
26.           "=m"((*resultVector).x2),
27.           "=m"((*resultVector).x3)
28.
29.         : "m"(firstVector.x0),
30.           "m"(firstVector.x1),
31.           "m"(firstVector.x2),
32.           "m"(firstVector.x3),
33.           "m"(secondVector.x0),
34.           "m"(secondVector.x1),
35.           "m"(secondVector.x2),
36.           "m"(secondVector.x3)
37.
38.     );
39. }
```

Listing 2. Funkcja dodająca dwa wektory w typie obliczeń SISD.

W funkcji tej wykonywane są powtarzalne mechanizmy, dlatego omówię tylko pierwszy z nich. W linii 4 zmienna wejściowa z linii nr. 29 ładowana jest na osobny stos FPU, następnie

zmienna ta dodawana jest z wartością wejściową wprowadzaną w linii nr. 33, a wynik umieszczany jest w rejestrze st0. Następnie wynik operacji dodawania przenoszony jest z st0 do zmiennej x0, wektora wynikowego w linii 6. Reszta operacji wykonywana jest w sposób analogiczny, dla pozostałych zmiennych wchodzących w skład danych struktur.

5. Opis uruchomienia programu

Wykorzystane przeze mnie komendy potrzebne do kompilowania, uruchomienia i debugowania programu to w kolejności.

- „gcc -m32 -ggdb Lab4.c” – do kompilacji całego programu.
- „./a.out” – do uruchomienia gotowego programu.
- „gdb a.out” – do debugowania tworzonych programów.

Dodatkowo w trakcie debugowania wykorzystywałem takie polecenia jak:

- „x/gf &liczba1” – do wyświetlenia liczby, zapisanej pod danym adresem w pamięci, w systemie dziesiętnym,
- „i r xmm0” lub „i r sse” – do podejrzenia wartości znajdujących się w rejestrach oznaczonych jako „XMM”.
- „disass” – do dezasemblacji części kodu napisanej w języku C.

A także inne takie jak „b nr linii”, „run”, „stepi”,”c” do kontroli nad przebiegiem debugowania programu.

Program w postaci makefile prezentuję poniżej.

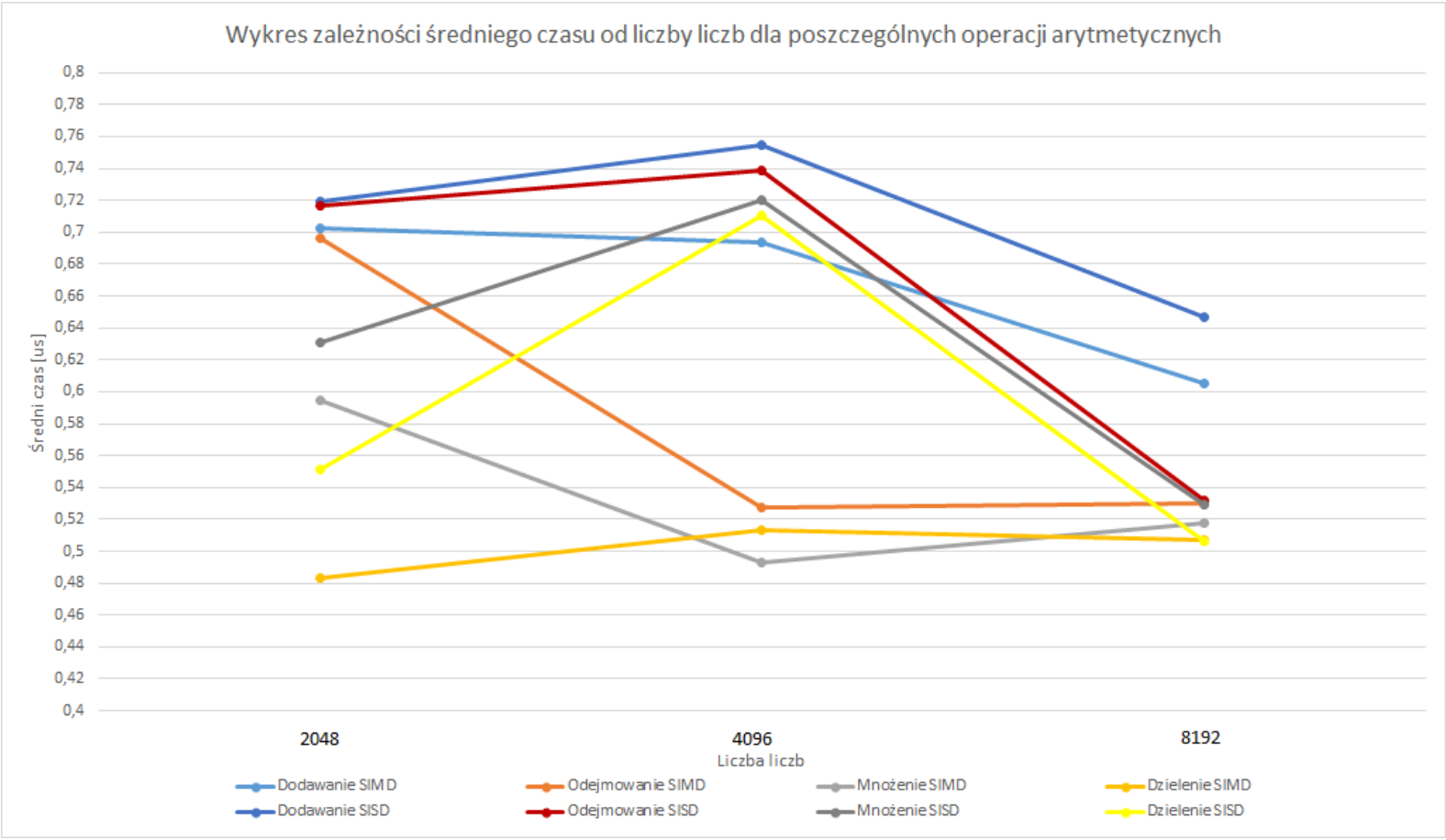
```
All: Lab4.c
	gcc -o Lab4 Lab4.c
```

Listing 3. Zawartości pliku makefile.

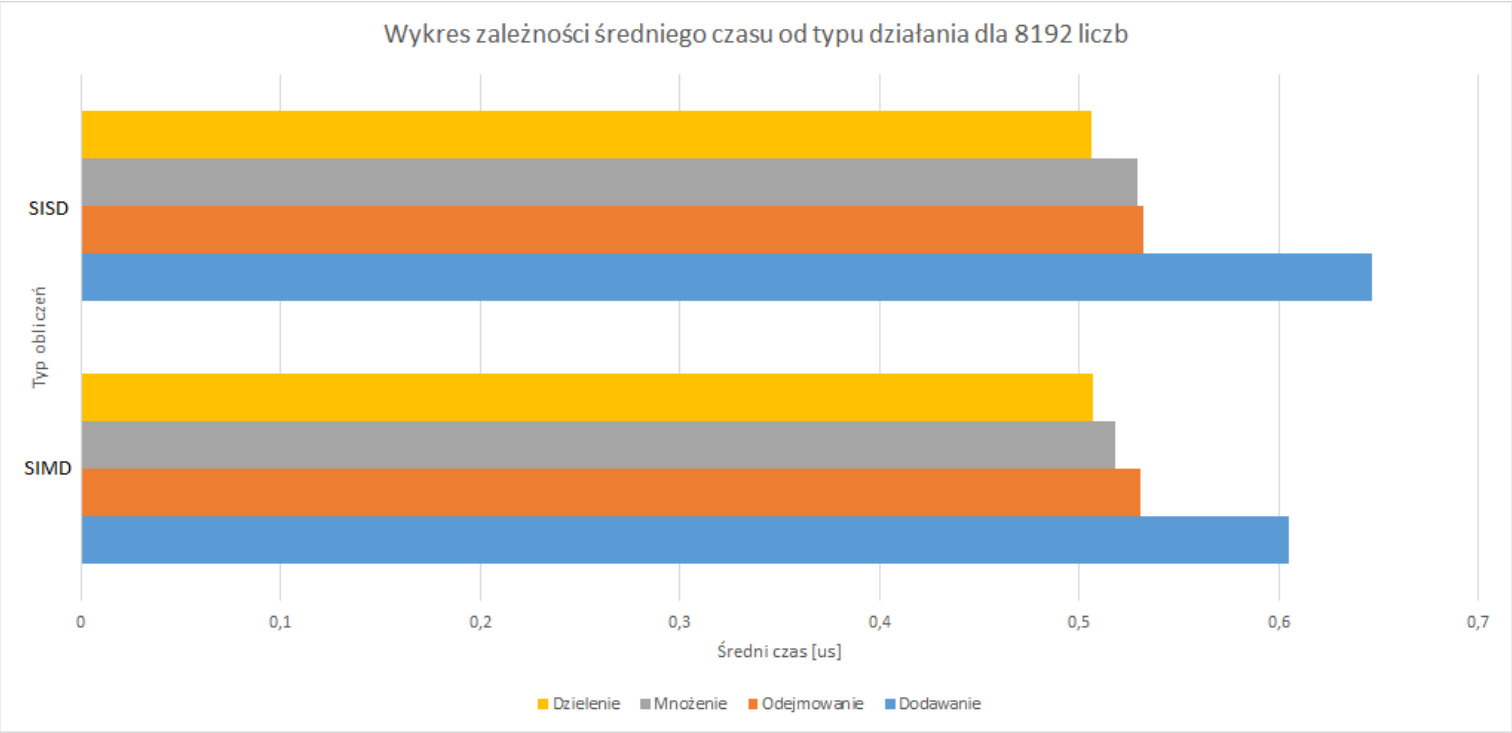
6. Wykresy

Niezbędne wykresy na podstawie otrzymanych wyników prezentuję poniżej. Pomiary zostały wykonane na komputerze o poniższej specyfikacji:

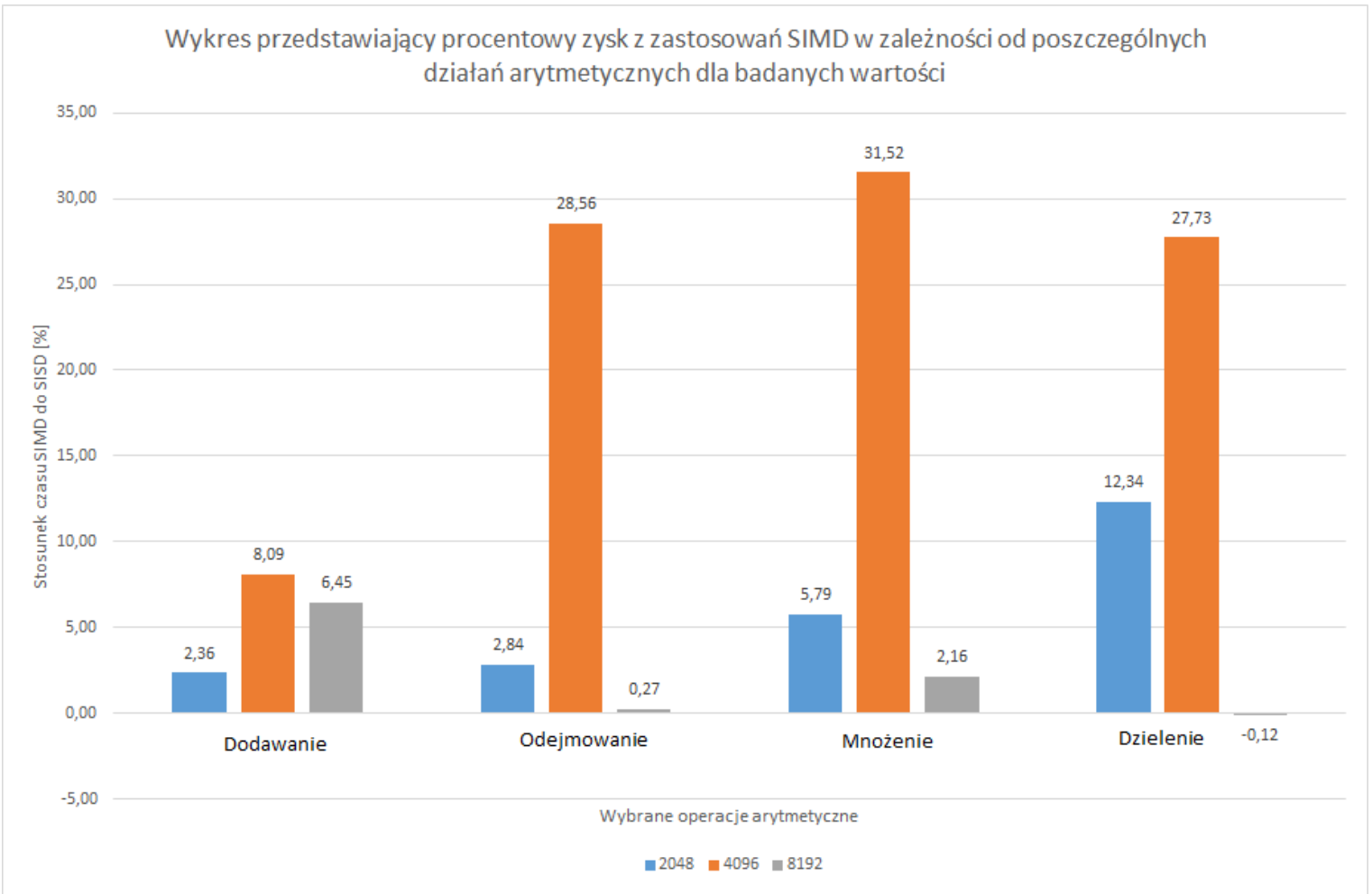
- Procesor sześciordzeniowy - Intel Core i5 8600K - o częstotliwości taktowania 3.60GHz.
- Pamięć RAM typu DDR4 o łącznej pamięci 8GB i częstotliwości 2666MHz.



Wykres 1. Przedstawiający zależność średniego czasu od liczby liczb dla poszczególnych operacji arytmetycznych



Wykres 2. Przedstawia wykres zależności średniego czasu od typu działania dla 8192 liczb.



Wykres 3. Przedstawia procentowy zysk z zastosowania SIMD w zależności od poszczególnych działań arytmetycznych dla danych wartości

7. Wnioski

Na podstawie wykresów przedstawionych w rozdziale nr 6. Przedstawić mogę następujące wnioski.

Po pierwsze na podstawie danych zawartych na wykresach można wyraźnie zaobserwować różnice w prędkości działania operacji wykonywanych przy pomocy typu SIMD. Jest to efekt spodziewany, ponieważ wykonywanie operacji równolegle, powinno być znacznie szybsze niż wyliczanie tych samych wartości jeden za drugim. W większości przypadków zgadza się to również z otrzymanymi wynikami. Wyjątkiem stanowi jedynie porównanie dzielenia dla 8192 liczb, ponieważ jest to jedyny przypadek w którym dzielenie z wykorzystaniem SIMD jest wolniejsze niż SISD. Najprawdopodobniej jest to błąd systematyczny wynikający z obciążenia komputera przez procesy systemowe w momencie zbierania pomiarów.

Po drugie warto zwrócić uwagę na wykres nr 3 dla danych zebrach z obliczeń przeprowadzonych na 4096 liczbach. Jest to o tyle ciekawy przypadek, iż wykazuje on największy procentowy zysk z zastosowania typu SIMD do obliczeń. Może być to spowodowane jak w powyższym przypadku najpewniej, obciążeniem komputera przez procesy systemowe, których wyłącznie było niemożliwe. Dodatkowym hipotetycznym argumentem na wyjaśnienie zaistniałej sytuacji może być wrażliwość na dane wejściowe w przypadku obliczeń typu SISD, ponieważ w przypadku generowania pożyczek lub przeniesień, musiałyby zostać wykonane dodatkowe operacje (na pojedynczej zmiennej), które mogłyby w taki sposób przedłużyć czas wykonywania obliczeń.

Podsumowując, uważam, że wykonane zadanie, a następna analiza otrzymanych wyników pozwoliła na przedstawienie zalet związanych ze stosowania typu obliczeń SIMD w programach. Pomimo kilku przypadków niewielkiego zysku czasowego (lub nawet spadku) przy obliczeniach wykonywanych w bardziej złożonym kodzie zysk ten może przynieść naprawdę spore korzyści w szybkości działania całego programu. Dodatkowo, na podstawie porównania listingu 1 i 2, sądzę że dodatkową zaletą wykorzystania SIMD jest jego zwięzłość oraz większa czytelność kodu.

8. Literatura

[1] Użycie języka assembler wewnątrz C - https://www.codeproject.com/Articles/15971/Using-Inline-Assembly-in-C-C, https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html#Extended-Asm

[2] Instrukcje SSE, oraz opis ich działania - https://en.wikibooks.org/wiki/X86_Assembly/SSE,

http://students.mimuw.edu.pl/~zbyszek/asm/pl/instrukcje-sse.html